Container Devops in 3 Weeks

Agenda

Poll Question

What is your experience with DevOps

- What is DevOps?
- None
- Just starting
- Reasonable
- Advanced

Poll Question

Which of the following topics do you feel already confident with? (select all that apply)

- 12 Factor Apps
- Understanding DevOps
- Working with Containers
- Kubernetes basics
- Kubernetes intermediate
- OpenShift basics
- OpenShift intermediate

Poll Question

- Where are you from?
- India
- Asia (not India)
- USA or Canada
- Central America
- South America
- Africa
- Netherlands
- Europe
- Australia/Pacific

Course Overview

- On day 1, you'll learn about DevOps fundamentals. It has significant amount of lectures, and you'll learn how to work with GitHub and essential DevOps tools. We'll also start exploring containers
- On day 2, we'll further explore containers, the preferred way of offering access to appplications in a DevOps world. A strong focus is on managing container images the DevOps way, and we'll start exploring Kubernetes and OpenShift
- On day 3, you'll learn how to work the DevOps way with Kubernetes, the perfect tool to build container based microservices and decouple site-specific information from the code you want to distribute

Course Objectives

- In this course, you will learn about DevOps and common DevOps solutions
- You will learn how to apply these solutions in Orchestrated Containerized IT environments
- We'll zoom into the specific parts, but in the end the main goal is to bring these parts together, allowing you to make DevOps work more efficient by working with containers

Day 1 Agenda

- Understanding DevOps
- Understanding 12 Factor App Development
- Using Git
- Using CI/CD
- Understanding Microservices
- Using Containers in Microservices
- Getting started with Ansible
- Running Containers in Docker or Podman

Day 2 Agenda

- Managing Container Images
- Triggering Image Builds from Git Repositories
- Managing Container Storage
- Understanding Kubernetes
- Using Kubernetes and OpenShift
- Exploring Basic Kubernetes and OpenShift Skills

Day 3 Agenda

- Using Kubernetes the DevOps way
- Exposing Applications
- Configuring Application Storage
- Implementing Decoupling in Kubernetes
- Understanding Helm Charts, Operators and Custom Resources
- Building OpenShift Applications from Git Source Code

How this course is different

- Topics in this course are further exploring in other courses I'm teaching
    - Containers in 4 Hours
    - Kubernetes in 4 Hours
    - CKAD, CKA
    - Ansible in 4 Hours
    - Getting Started with OpenShift
    - EX180, EX280
- This course is different, as its focus is on using these platforms as tools in DevOps
- The orientation in this course is on DevOps techniques

Container Devops in 3 Weeks

Day 1

Day 1 Agenda

- Understanding DevOps
- Understanding 12 Factor App Development
- Using Git
- Using CI/CD
- Understanding Microservices
- Using Containers in Microservices
- Getting started with Ansible
- Running Containers in Docker or Podman

Understanding DevOps

Understanding DevOps

- In DevOps, Developers and Operators work together on implementing new software and updates to software in the most efficient way
- The purpose of DevOps is to reduce the time between committing a change to a system and the change being placed in production
- DevOps is Microservices-oriented by nature, as multiple smaller project are easier to manage than one monolithic project
- In DevOps, CI/CD is commonly implemented, using anything from simple GitHub repositories, up to advanced CI/CD-oriented software solutions such as Jenkins and OpenShift

DevOps Key Components

- Configuration as Code
- The DevOps Cycle
- Microservices
- The 12-factor application

Configuration as Code

- In the DevOps way of working, Configuration as code is the common approach
- Complex commands are to be avoided, use manifest files containing the desired configuration instead
- YAML is a common language to create these manifest files
- YAML is used in different DevOps based solutions, including Kubernetes and Ansible

Understanding The DevOps Cycle

This is the framework for this course

- Coding: source code management tools
- Building: continuous integration tools
- Testing: continuous testing tools
- Packaging: packaging tools
- Releasing: release automation
- Configuring: configuration management tools
- Monitoring: applications monitoring

Understanding Microservices

Understanding Microservices

- Microservices define an application as a collection of loosely coupled services
- Each of these services can be deployed independently
- Each of them is independently developed and maintained
- Microservices components are typically deployed as containers
- Microservices are a replacement of monolithic applications
- Microservices are often implemented as containers that are orchestrated by Kubernetes

Microservices benefits

- When broken down in pieces, applications are easier to build and maintain
- Smaller pieces are easier to understand
- Developers can work on applications independently
- Smaller components are easier to scale
- One failing component doesn't necessarily bring down the entire application

Understanding 12-Factor Apps

What is the 12-Factor App

- The 12-factor app is a development methodology for building apps that
  - Use declarative formats
  - Offer maximum portability
  - Are suitable for deployment on cloud platforms
  - Enable continuous deployment, which minimizes divergence
  - Allows for easy scaling of applications
- 12-factor app based DevOps explains why orchestrating containerized workloads in Kubernetes is essential
- See 12factor.net for more details

The 12 factors (1)

- I. Codebase: One codebase, tracked in revision control, many deployes: Git, declarative code, Dockerfile
- II. Dependencies: Explicitely declare and isolate dependencies: Kubernetes Probes, init containers
- III. Config: Store config in the environment: ConfigMap
- IV: Backing Services: Treat Backing services as attached resources: Service Resources, pluggable networking
- V: Build, release, run: Strictly separate build and run stages: CI/CD, S2I, Git branches, Helm
- VI: Processes: Execute the app as one or more stateless processes: Microservices, Linux kernel namespaces

The 12 factors (2)

- VII: Port Binding: Export services via port binding: K8s Services, Routes
- VIII: Concurrency: Scale out via the process model: K8s ReplicaSets
- IX: Disposability: Maximize robustness with fast startup and graceful shutdown: K8s probes
- X: Dev/prod parity: Keep development, staging and production as similar as possible: Containers
- XI: Logs: Treat logs as event streams: Logs stored in the orchestration layer
- XII: Admin processes: Run admin/management tasks as one-off processes: Ansible Playbooks, Kubernetes Jobs

Container Devops in 4 Weeks

Coding: Using Git

Using Git in a Microservices Environment

- Git can be used for version control and cooperation between different developers and teams
- Using Git makes it easy to manage many changes in an effective way
- Different projects in a Microservice can have their own Git repository
- For that reason, Git and Microservices are a perfect match

Using Git

- Git is typically offered as a web service
- GitHub and GitLab are commonly used
- Alternatively, private Git repositories can be used

Understanding Git

- Git is a version control system that makes collaboration easy and effective
- Git works with a repository, which can contain different development branches
- Developers and users can easily upload as well as download new files to and from the Git repository
- To do so, a Git client is needed
- Git clients are available for all operating systems
- Git servers are available online, and can be installed locally as well
- Common online services include GitHub and GitLabs

Git Client and Repository

- The Git repository is where files are uploaded, and shared with other users
- Individual developers have a local copy of the Git repository on their computer and use the Git client to upload and download to and from the repository
- The organization of the Git client lives in the .git directory, which contains several files to maintain the status

Understanding Git Workflow

- To offer the best possible workflow control, A Git repository consists of three trees maintained in the Git-managed directory
  - The *working directory* holds the actual files
  - The *Index* acts as a staging area
  - The *HEAD* points to the last commit that was made

Applying the Git Workflow

- The workflow starts by creating new files in the working directory
- When working with Git, the **git add** command is used to add files to the index
- To commit these files to the head, use **git commit -m "commit message"**
- Use **git add origin https://server/reponame** to connect to the remote repository
- To complete the sequence, use **git push origin master**. Replace "master" with the actual branch you want to push changes to

Creating a GitHub Repository

- Create the repository on your GitHub server
- Set your user information
  - **git config --global user.name “Your Name”**
  - **git config --global user.email “you@example.com”**
- Create a local directory that contains a README.md file. This should contain information about the current repository
- Use **git init** to generate the Git repository metadata
- Use **git add <filenames>** to add files to the staging area
- From there, use **git commit -m "commit message"** to commit the files. This will commit the files to HEAD, but not to the remote repository yet
- Use **git remote add origin https://server/reponame**
- Push local files to remote repository: **git push -u origin master**

Using Git Repositories

- Use **git clone https://gitserver/reponame** to clone the contents of a remote repository to your computer
- To update the local repository to the latest commit, use **git pull**
- Use **git push** to send local changes back to the Git server (after using **git add** and **git commit** obviously)

Uploading Changed Files

- Modified files need to go through the staging process
- After changing files, use **git status** to see which files have changed
- Next, use **git add** to add these files to the staging area; use **git rm <filename>** to remove files
- Then, commit changes using **git commit -m "minor changes"**
- Synchronize, using **git push origin master**
- From any client, use **git pull** to update the current Git clone

Understanding Branches

- Branches are used to develop new features in isolation from the main branch
- The master branch is the default branch, other branches can be manually added
- After completion, merge the branches back to the master

Using Branches

- Use **git checkout -b dev-branch** to create a new branch and start using it
- Use **git push origin dev-branch** to push the new branch to the remote repository
- Use **git checkout master** to switch back to the master
- Use **git merge dev-branch** to merge the dev-branch back into the master
- Delete the branch using **git branch -d dev-branch**

Lab: Using Git

- Got to https://github.com, and create an account if you don't have an account yet
- Create a new Git repository from the website
- From a Linux client, create a local directory with the name of the Git repository
- Use the following commands to put some files in it
  - echo "new git repo" >README.md
  - git init
  - git add *
  - git status
  - git commit -m "first commit"
  - git remote add origin https://github.com/yourname/yourrepo
  - git push -u origin master

Container Devops in 4 Weeks

Understanding CI/CD

What is CI/CD

- CI/CD is Continuous integration and continuous deliver/continuous deployment
- It's a core Devops element that enforces automation in building, testing and deployment of applications
- The CI/CD pipeline is the backbone of modern DevOps operations
- In CI, all developers merge code changes in a central repository multiple times a day
- CD automates the software release process based on these frequent changes
- To do so, CD includes automated infrastructure provisioning and deployment

Understanding CI/CD pipelines

- The Ci/CD pipeline automates the software delivery process
- It builds code, runs tests (CI) and deployes a new version of the application (CD)
- Pipelines are automated so that errors can be reduced
- Pipelines are a runnable specification of the steps that a developer needs to perform to deliver a new version of a software product
- A CI/CD pipeline can be used as just a procedure that describes how to get from code to running software
- CI/CD pipelines can also be automated using software like Jenkins or OpenShift

Understanding Stages of Software Release

- 1: From source to Git: git push
- 2: From Git to running code: docker build, make
- 3: Testing: smoke test, unit test, integration test
- 4: Deployment: staging, QA, production

Source Stage

- Source code ends up in a repository
- Developers need to use **git push** or something to get their software into the repository
- The pipeline run is triggered by the source code repository

Build Stage

- The source code is converted into a runnable instance
- Source code written in C, Go or Java needs to be compiled
- Cloud-native software is deployed by using container images
- Failure to pass the build stage indicates there's a fundamental problem in either the code or the generic CI/CD configuration

Test Stage

- Automated testing is used to validate code correctness and product behavior
- Automated tests should be written by the developers
- Smoke tests are quick sanity checks
- End-to-end tests should test the entire system from the user point of view
- Typically, test suites are used
- Failure in this stage will expose problems that the developers didn't foresee while writing their code

Deploy Stage

- In deployment, the software is first deployed in a beta or staging environment
- After is passes the beta environment successfully, it can be pushed to the production environment for end users
- Deployment can be a continuous process, where different parts of a microservice are deployed individually and can automatically be approved and commited to the master branch for production

Benefits of using pipelines

- Developers can focus on writing code and monitoring behavior of their code in production
- QA have access to the latest version of the system at any time
- Product updates are easy
- Logs of all changes are always available
- Rolling back to a previous version is easy
- Feedback can be provided fast

Container Devops in 4 Weeks

Configuration Management: Using Ansible in DevOps

What is Ansible?

- Ansible is a Configuration Management tool
- It can be used to manage Linux, Windows, Network Devices, Cloud, Docker and more
- The Control node runs the Ansible software, which is based on Python
- The Control node reaches out to the managed nodes to compare the current state with the desired state
- Desired state is defined in Playbooks, that are written in YAML

Why is Ansible DevOps?

- Ansible is Configuration as Code

Setting up a simple Ansible Environment

- On control hosts
  - Use CentOS 8.x
  - Enable EPEL repository
  - Enable host name resolving for all managed nodes
  - Generate SSH keys and copy over to managed hosts
  - Install Ansible software
  - Create an inventory file
- On managed hosts
  - Ensure Python is installed
  - Enable (key-based) SSH access
  - Make sure you have a user with (passwordless) sudo privileges

Lab: Setting up Ansible

- On the Ubuntu 20.04 LTS managed hosts
  - **sudo apt-install openssh-server**
- On the CentOS 8.x control host
  - **sudo dnf install epel-release**
  - **sudo dnf install –y ansible**
  - **sudo sh –c 'echo <your.ip.addr.ess> ubuntu.example.com ubuntu >> /etc/hosts'**
  - **ssh-keygen**
  - **ssh-copy-id ubuntu**
  - **echo ubuntu >> inventory**
  - **ansible ubuntu –m ping –i inventory –u student**

Using Ad-Hoc Commands

- Ansible provides 3000+ different modules
- Modules provide specific functionality and run as Python scripts on managed nodes
- Use **ansible-doc -l** for a list of all modules
- Modules can be used in ad-hoc commands:
  - **ansible ubuntu -i inventory -u student -b -K -m user -a "name=linda"**
  - **ansible ubuntu -i inventory -u student -b -K -m package -a "name=nmap"**

Using ansible.cfg

- While using Ansible commands, command line options can be used to provide further details
- Alternatively, use ansible.cfg to provide some standard values
- An example ansible.cfg is in the Git repository at https://github.com/sandervanvugt/devopsinfourweeks

Using Playbooks

- Playbooks provide a DevOps way for working with Ansible
- In a playbook the desired state is defined in YAML
- The **ansible-playbook** command is used to compare the current state of the managed machine with the desired state, and if they don't match the desired state is implemented
- **ansible-playbook -i inventory -u student -K my-playbook.yaml**

Understanding Containers

Understanding Containers

- A container is a running instance of a container image that is fetched from a registry
- An image is like a smartphone App that is downloaded from the AppStore
- It's a fancy way of running an application, which includes all that is required to run the application
- A container is NOT a virtual machine
- Containers run on top of a Linux kernel, and depend on two important kernel features
  - Cgroups
  - Namespaces

Understanding Container History

- Containers started as chroot directories, and have been around for a long time
- Docker kickstarted the adoption of containers in 2013/2014
- Docker was based on LXC, a Linux native container alternative that had been around a bit longer

Understanding Container Solutions

- Containers run on top of a container engine
- Different Container engines are provided by different solutions
- Some of the main solutions are:
  - Docker
  - Podman
  - LXC/LXD
  - systemd-nspawn

Understanding Container Types

- System containers are used as the foundation to build your own application containers. They are not a replacement for a virtual machine
- Application containers are used to start just one application. Application containers are the standard
- To run multiple connected containers, you need to create a microservice. Use docker-compose or Kubernetes Pods to do this in an efficient way

Using Ansible to Setup a Docker Environment

Demo: Using Ansible to Setup Docker

- Make sure you have setup the Ubuntu 20.04 workstation for management by Ansible
- Use **ansible-playbook -u student -K -i inventory ansible-ubuntu.yml** to set up the Ubuntu host
- On Ubuntu, log out and log in as your user **student**
- Use **docker run hello-world**

Container Devops in 4 Weeks

Running Containers in Docker and Podman

Podman or Docker?

- Red Hat has changed from Docker to Podman as the default container stack in RHEL 8
- Docker is no longer supported in RHEL 8 and related distributions
- Even if you can install Docker on top of RHEL 8, you shouldn't do it as it will probably break with the next software update
- Podman is highly compatible with Docker
- By default, Podman runs rootless containers, which have no IP address and cannot bind to privileged ports
- Both Docker as Podman are based on OCI standards
- For optimal compatibility, install the **podman-docker** package

Demo: Running Containers

- docker run ubuntu
- docker ps
- docker ps -a
- docker run -d nginx
- docker ps
- docker run -it ubuntu sh; Ctrl-p, Ctrl-q
- docker inspect ubuntu
- docker rm ubuntu
- docker run --name webserver --memory="128m" -d -p 8080:80 nginx
- curl localhost:8080

Container Devops in 3 Weeks

Day 2

Day 2 Agenda

- Managing Container Images
- Triggering Image Builds from Git Repositories
- Managing Container Storage
- Understanding Kubernetes
- Using Kubernetes and OpenShift
- Exploring Basic Kubernetes and OpenShift Skills

Managing Container Images

Understanding Images

- A container is a running instance of an image
- The image contains application code, language runtime and libraries
- External libraries such as libc are typically provided by the host operating system, but in container is included in the image
- While starting a container it adds a writable layer on the top to store any changes that are made while working with the container
- These changes are ephemeral
- Container images are highly compatible, and either defined in Docker or in OCI format

Getting Container Images

- Containers are normally fetched from registries
- Public registries such as https://hub.docker.com are available
- Red hat offers https://quay.io as a registry with more advanced CI features offered
- Alternatively, private registries can easily be created
- Use Dockerfile to create custom images

Fetching Images from Registries

- By default, Docker fetches containers from Docker Hub
- In Podman, the /etc/containers/registries.conf file is used to specify registry location
- Alternatively, the complete path to an image can be used to fetch it from a specific registry: **docker pull localhost:5000/fedora:latest**

Understanding Image Tags

- Normally, different versions of images are available
- If nothing is specified, the latest version is pulled
- Use tags to pull a different version: **docker pull nginx:1.14**

Demo: Managing Container Images

- Explore **https://hub.docker.com**
- **docker search mariadb** will search for the mariadb image
- **docker pull mariadb**
- **docker images**
- **docker inspect mariadb**
- **docker image history mariadb**
- **docker image rm mariadb**

Container Devops in 4 Weeks

Using Dockerfile

Understanding Dockerfile

- Dockerfile is a way to automate container builds
- It contains all instructions required to build a container image
- So instead of distributing images, you could just distribute the Dockerfile
- Use **docker build .** to build the container image based on the Dockerfile in the current directory
- Images will be stored on your local system, but you can direct the image to be stored in a repository
- Tip: images on hub.docker.com have a link to Dockerfile. Read it to understand how an image is build using Dockerfile!

Using Dockerfile Instructions

- FROM: identifies the base image to use. This must be the first instruction in Dockerfile
- MAINTAINER: the author of the image
- RUN: executes a command while building the container, it is executed before the container is run and changes what is in the resulting container
- CMD: specifies a command to run when the container starts
- EXPOSE: exposes container ports on the container host
- ENV: sets environment variables that are passed to the CMD
- ADD: copies files from the host to the container. By default files are copied from the Dockerfile directory
- ENTRYPOINT: specifies a command to run when the container starts

Using Dockerfile Instructions

- VOLUME: specifies the name of a volume that should be mounted from the host into the container
- USER: identifies the user that should run tasks for building this container, use for services to run as a specific user
- WORKDIR: set the current working directory for commands that are running from the container

Understanding ENTRYPOINT and CMD

- Both ENTRYPOINT and CMD specify a command to run when the container starts
- CMD specifies the command that should be run by default after starting the container. You may override that, using **docker run mycontainer othercommand**
- ENTRYPOINT can be overridden as well, but it's more work: you need **docker run --entrypoint mycommand mycontainer** to override the default command that is started
- Best practice is to use ENTRYPOINT in situations where you wouldn't expect this default command to be overridden

ENTRYPOINT and CMD Syntax

- Commands in ENTRYPOINT and COMMAND can be specified in different ways
- The most common way is the Exec form, which is shaped as **<instruction> ["executable", "arg1", "arg2"]**
- The alternative is to use Shell form, which is shaped as **<instruction> <command>**
- While shell form seems easier to use, it runs <command> as an argument to /bin/sh, which may lead to confusion

Demo: Using a Dockerfile

- Dockerfile demo is in https://github.com/devopsinfourweeks/dockerfile
- Use **docker build -t nmap .** to run it from the current directory
- Tip: use **docker build --no-cache -t nmap .** to ensure the complete procedure is performed again if you need to run again
- Next, use **docker run nmap** to run it

Lab: Working with Dockerfile

- Create a Dockerfile that deploys an httpd web server that is based on the latest Fedora container image. Use a sample file index.html which contains the text "hello world" and copy this file to the /var/www/html directory. Ensure that the following packages are installed: nginx curl wget
- Use the Dockerfile to generate the image and test its working

Publishing on Docker Hub

Demo: Creating an autobuild Repo on Docker Hub

- Access https://github.com
- If required, create an account and log in
- Click **Create Repository**
- Enter the name of the new repo; e.g. **devops** and set to **Public**
- Check **Settings** > **Webhooks**. Don't change anything, but check again later

Demo: Creating an autobuild Repo on Docker Hub

- On a Linux console, create the local repository
  - `mkdir devops`
  - `echo "hello" >> README.md`

  -
    ```
    cat > Dockerfile <<EOF
    FROM busybox
    CMD echo "Hello world!"
    EOF
    ```
  - `git init`
  - `git add *`
  - `git commit -m "initial commit"`
  - `git remote add origin https://github.com/yourname/devops.git`
  - `git push -u origin master`

Demo: Creating an autobuild Repo on Docker Hub

- Access https://hub.docker.com
- If required, create an account and log in
- Click **Create Repository**
- Enter the name of the new repo; e.g. **devops** and set to **Public**
- Under Build Settings, click Connected and enter your GitHib "organization" as well as a repository

Demo: Creating an autobuild Repo on Docker Hub

- Still from hub.docker.com: Add a Build Rule that sets the following:
  - Source: Branch
  - Source: master
  - Docker Tag: latest
  - Dockerfile location: Dockerfile
  - Build Context: /
- Check Builds > Build Activity to see progress
- Once the build is completed successfully, from a terminal user **docker pull yourname/devops:latest** to pull this latest image (may take a minute to synchronize)
- On GitHub, check Settings > Webhooks for your repo - settings should be automatically added

Demo: Creating an autobuild Repo on Docker Hub

- From the Git repo on the Linux console: edit the Dockerfile and add the following line: MAINTAINER yourname your@mailaddress.com
- **git status**
- **git add ***
- **git commit -m "minor update"**
- **git push -u origin master**
- From Docker hub: Check your repository > Builds > Build Activity. You'll see that a new automatic build has been triggered (should be fast)

Container Devops in 4 Weeks

Managing Container Storage

Configuring Storage

- To work with storage, bind mounts and volumes can be used
- A bind mount provides access to a directory on the Docker host
- Volumes exist outside of the container spec, and as such outlive the container lifetime
- Volumes offer an option to use other storage types as well
- Within Docker-CE, volume types are limited to **local** and **nfs**
- In Kubernetes or Docker Swarm more useful storage types are provided

Demo: Using an NFS-based Volume -1

- sudo apt install nfs-server nfs-common
- sudo mkdir /nfsdata
- sudo vim /etc/exports
    - /nfsdata   *(rw,no_root_squash)
- sudo chown nobody:nogroup /nfsdata
- sudo systemctl restart nfs-kernel-server
- showmount -e localhost

Demo: Using an NFS-based Volume -1

- docker volume create --driver local --opt type=nfs --opt o=addr=127.0.0.1,rw --opt device=:/nfsdata nfsvol
- docker volume ls
- docker volume inspect nfsvol
- docker run -it --name nfstest --rm --mount source=nfsvol,target=/data nginx:latest /bin/sh
- touch /data/myfile; exit
- ls /nfsdata

Lab: Managing Volumes

- Use **docker volume create myvol** to create a volume that uses local storage as its backend
- Inspect the volume to see what it's doing with files that are created
- Mount this volume in an nginx container in the directory /data. Create a file in this directory and verify this file is created on the local volume storage backend

Working with Volumes

- **docker volume create myvol** creates a simple volume that uses the local file system as the storage backend
- **docker volume ls** will show the volume
- **docker volume inspect my-vol** shows the properties of the volume
- **docker run -it --name voltest --rm --mount source=myvol,target=/data nginx:latest /bin/sh** will run a container and attach to the running volume
- From the container, use **cp /etc/hosts /data; touch /data/testfile; ctrl-p, ctrl-q**
- **sudo -l; ls /var/lib/docker/volumes/myvol/_data/**
- **docker run -it --name voltest2 --rm --mount source=myvol,target=/data nginx:latest /bin/sh**
- From the second container: **ls /data; touch /data/newfile; ctrl-p, ctrl-q**

Using Docker Compose

Understanding Docker Compose

- Docker Compose uses the declarative approach to start Docker containers, or Microservices consisting of multiple Docker containers
- The YAML file is used to include parameters that are normally used on the command line while starting a Docker container
- To use it, create a **docker-compose.yml** file in a directory, and from that directory run the **docker-compose up -d** command
- Use **docker-compose down** to remove the container

Demo: Bringing up a Simple Nginx Server

- Use the simple-nginx/docker-compose.yml file from https://github.com/sandervanvugt/devopsinfourweeks
- **cd simple-nginx**
- **docker-compose up -d**
- **docker ps**

Demo: Bringing up a Microservice

- Use the wordpress-mysql/docker-compose.yml file from https://github.com/sandervanvugt/devopsinfourweeks
- **cd wordpress-mysql**
- **docker-compose up -d**
- **docker ps**

Lab: Using Docker Compose

- Start an nginx container, and copy the /etc/nginx/conf.d/default.conf configuration file to the local directory ~/nginx-conf/
- Use Docker compose to deploy an application that runs Nginx. Expose the application on ports 80 and 443 and mount the configuration file by using a volume that exposes the ~/nginx-conf directory

Container Devops in 4 Weeks

Understanding Kubernetes

Understanding Kubernetes

- Kubernetes offers enterprise features that are needed in a containerized world
  - Scalability
  - Availability
  - Decoupling between static code and site specific data
  - Persistent external storage
  - The flexibility to be used on premise or in cloud
- Kubernetes is the de facto standard and currently there are no relevant competing products

Options for Using Kubernetes

- Managed in Cloud
- Minikube
- In Docker Desktop
- AiO
- As a distribution: OpenShift (or others)
- If using OpenShift: CodeReady Containers

Installing Kubernetes

- In cloud, managed Kubernetes solutions exist to offer a Kubernetes environment in just a few clicks
- On premise, administrators can build their own Kubernetes cluster using **kubeadm**
- For testing, **minikube** can be used
- In this course we'll build an AiO on prem K8s cluster
- See "Setup Guide.pdf" in https://github.com/sandervanvugt/devopsinfourweeks for a detailed procedure description

Installing an AiO on-prem Cluster - 1/4

- Install some packages
  - **yum install git vim bash-completion**
- As ordinary user with sudo privileges, clone the course Git repository
  - **git clone https://github.com/sandervanvugt/devopsinfourweeks**
- Run the setup scripts:
  - **cd /devopsinfourweeks**
  - **./setup-docker.sh**
  - **./setup-kubetools.sh**
- In a root shell, install a Kubernetes master node
  - **kubeadm init --pod-network-cidr=10.10.0.0/16**

Installing an AiO on-prem Cluster - 2/4

- In a user shell, set up the kubectl client:
  - **mkdir -p $HOME/.kube**
  - **sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config**
  - **sudo chown $(id -un):$(id -un) .kube/config**

Installing an AiO on-prem Cluster - 3/4

- In a user shell, set up the Calico networking agent
  - **kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml**
  - **wget https://docs.projectcalico.org/manifests/custom-resources.yaml**
  - You now need to define the Pod network, which by default is set to 192.168.0.0/24, which in general is a bad idea. I suggest setting it to 10.10.0.0 - make sure this address range is not yet used for something else!
  - **sed -i -e s/192.168.0.0/10.10.0.0/g custom-resources.yaml**
  - **kubectl create -f custom-resources.yaml**
  - **kubectl get pods -n calico-system**: wait until all pods show a state of Ready, this can take about 5 minutes!

Installing an AiO on-prem Cluster - 4/4

- By default, user Pods cannot run on the Kubernetes control node. Use the following command to remove the taint so that you can schedule nodes on it:
  kubectl taint nodes --all node-role.kubernetes.io/master-
- Type **kubectl get all** to verify the cluster works.
- Use **kubectl create deployment nginx --image=nginx** to verify that you can create applications in Kubernetes

Getting Started with CodeReady Containers

Understanding CodeReady Containers

- CodeReady Containers (CRC) is a free all-in-one OpenShift solution
- You need a free Red Hat developer account
- CodeReady Containers can be installed in different ways
  - On top of your current OS
  - Isolated in a Linux VM
- To prevent having conflicts with other stuff running on your computer, it's recommended to install in an isolated VM
- For other usage options, see here: https://developers.redhat.com/products/codeready-containers/overview

Installing CRC in an Isolated VM

- The VM needs the following
  - 12 GB RAM
  - 4 CPU cores
  - 40 GB disk
  - Support for nested virtualization
- Download the tar ball and the pull-secret
- Extract the tarball
- move the **crc** file to /usr/local/bin
- **crc setup**
- **crc start -p pull-secret -m 8192**

Container Devops in 4 Weeks

Running Applications in Kubernetes

Understanding Kubernetes Resources

- Kubernetes resources are defined in the APIs
- Use **kubectl api-resources** for an overview
- Kubernetes resources are extensible, which means that you can add your own resources

Understanding Kubernetes Key Resources

- Pod: used to run one (or more) containers and volumes
- Deployment: adds scalability and update strategy to pods
- Service: exposes pods for external use
- Persistent Volume Claim: connects to persistent storage
- ConfigMap: used to store site specific data separate from pods

Exploring **kubectl**

- **kubectl** is the main management interface
- Make sure that **bash-completion** is installed for awesome tab completion
- **source <(kubectl completion bash)**
- Explore **kubectl -h** at all levels of **kubectl**

Running Applications in Kubernetes

- **kubectl create deployment** allows you to create a deployment
- **kubectl run** allows you to run individual pods
- Individual pods (aka "naked pods") are unmanaged and should not be used
- **kubectl get pods** will show all running Pods in the current namespace
- **kubectl get all** shows running Pods and related resources in the current namespace
- **kubectl get all -A** shows resources in all namespaces

Troubleshooting Kubernetes Applications

- **kubectl describe pod <podname>** is the first thing to do: it shows events that have been generated while defining the application in the Etcd database
- **kubectl logs** connects to the application STDOUT and can indicate errors while starting application. This only works on running applications
- **kubectl exec -it <podname> -- sh** can be used to open a shell on a running application

Lab: Troubleshooting Kubernetes Applications

- Use **kubectl create deployment --image=busybox** to start a Busybox deployment
- It fails: use the appropriate tools to find out why
- After finding out why it fails, delete the deployment and start it again, this time in a way that it doesn't fail

Container Devops in 4 Weeks

Day 3

Day 3 Agenda

- Using Kubernetes the DevOps way
- Exposing Applications
- Configuring Application Storage
- Implementing Decoupling in Kubernetes
- Understanding Helm Charts, Operators and Custom Resources
- Building OpenShift Applications from Git Source Code

Poll Question

Have you attended the previous course days or watched its recordings?

- Day 1 only
- Day 2 only
- Day 1 and Day 2
- no

Container Devops in 4 Weeks

Managing Kubernetes the DevOps way

Pearson

Declarative versus Imperative

- In Imperative mode, the administrator uses command with command line options to define Kubernetes resources
- In Declarative mode, Configuration as Code is used by the DevOps engineer to ensure that resources are created in a consistent way throughout the entire environment
- To do so, YAML files are used
- YAML files can be written from scratch (not recommended), or generated: **kubectl create deployment mynginx --image=nginx --replicas=3 --dry-run=client -o yaml > mynginx.yaml**
- For complete documentation: use **kubectl explain <resource>.spec**

Exposing Applications

Understanding Application Access

- Kubernetes applications are running as scaled pods in the pod network
- The pod network is provided by the **kube-apiserver** and not reachable from the outside
- To expose access to applications, **service** resources are used

Demo: Exposing Applications

- kubectl create deploy mynginx --image=nginx --replicas=3
- kubectl get pods -o wide
- kubectl expose deploy mynginx --type=NodePort --port=80

Configuring Application Storage

Understanding K8s Storage Solutions

- Pod storage by nature is ephemeral
- Pods can refer to external storage to make it less ephemeral
- Storage can be decoupled by using Persistent Volume Claim (PCV)
- PVC addresses Persistent Volume
- Persistent Volume can be manually created
- Persistent Volume can be automatically provisioned using StorageClass
- StorageClass provides default storage in specific (cloud) environments
- Check **pv-pvc-pod.yaml** for an example

Container Devops in 4 Weeks

Implementing Decoupling in Kubernetes

Demo: Running MySQL

- kubectl run mymysql --image=mysql:latest
- kubectl get pods
- kubectl describe pod mymysql
- kubectl logs mymysql

Providing Variables to Kubernetes Apps

- In imperative way, the **-e** command line option can be used to provide environment variables to Kubernetes applications
- That's not very DevOps though, and something better is needed
- But let's verify that it works first: **kubectl run newmysql --image=mysql --env=MYSQL_ROOT_PASSWORD=password**
- Notice alternative syntax: **kubectl set env deploy/mysql MYSQL_DABASASE=mydb**

Understanding ConfigMaps

- ConfigMaps are used to separate site-specific data from static data in a Pod
  - Variables: **kubectl create cm variables --from-literal=MYSQL_ROOT_PASSWORD=password**
  - Config files: **kubectl create cm myconf --from-file=my.conf**
- Secrets are base64 encoded ConfigMaps
- Adressing the ConfigMap from a Pod depends on the type of ConfigMap
  - Use **envFrom** to address variables
  - Use **volumes** to mount ConfigMaps that contain files

Demo: Using a ConfigMap for Variables

- kubectl create cm myvars --from-literal=VAR1=goat --from-literal=VAR2=cow
- kubectl create -f cm-test-pod.yaml
- kubectl logs test-pod

Demo: Using a ConfigMap for Storage

- **kubectl create cm nginxconf --from-file nginx-custom-config.conf**
- **kubectl create -f nginx-cm.yml**
- Did that work? Fix it!
- **kubectl exec -it nginx-cm -- /bin/bash**
- **cat /etc/nginx/conf.d/default.conf**

Lab: Running MySQL the DevOps way

- Create a ConfigMap that stores all required MySQL variables
- Start a new mysql pod that uses the ConfigMap to ensure the availability of the required variables within the Pod

Comparing OpenShift to Kubernetes

Understanding OpenShift

- OpenShift is a Kubernetes distribution!
- Expressed in main functionality, OpenShift is a Kubernetes distribution where developer options are integrated in an automated way
  - Source 2 Image
  - Pipelines (currently tech preview)
  - More developed authentication and RBAC
- OpenShift adds more operators than vanilla Kubernetes
- OpenShift adds many extensions to the Kubernetes APIs

Container Devops in 4 Weeks

Running Kubernetes Applications in OpenShift

Running Applications in OpenShift

- Applications can be managed like in Kubernetes
- OpenShift adds easier to use interfaces as well
  - **oc new-app --docker-image=mariadb**
  - **oc set -h**
  - **oc adm -h**
- Managing a running environment is very similar
  - **oc get all**
  - **oc logs**
  - **oc describe**
  - **oc explain**
- etc.

Container Devops in 4 Weeks

Building OpenShift Applications from Git Source

Understanding S2I

- S2i allows you to run an application directly from source code
- **oc new-app** allows you to work with S2i directly
- S2i connects source code to an S2i image stream builder image to create a temporary builder pod that writes an application image to the internal image registry
- Based on this custom image, a deployment is created
- S2i takes away the need for the developer to know anything about Dockerfile and related items
- S2i also allows for continuous patching as updates can be triggered using web hooks

Understanding S2i Image Stream

- The image stream is offered by the internal image repository to provide different versions of images
- Use **oc get is -n openshift** for a list
- Image streams are managed by Red Hat through OpenShift. If a new version of the image becomes available, it will automatically trigger a new build of application code
- Custom image streams can also be integrated

Understanding S2i Resources

- ImageStream: defines the interpreter needed to create the custom image
- BuildConfig: defines all that is needed to convert source code into an image (Git repo, imagestream)
- DeploymentConfig/Deployment: defines how to run the container in the cluster; contains the Pod template that refers to the custom built image
- Service: defines how the application running in the deployment is exposed

Performing the S2i process

- **oc new-app php~https://github.com/sandervanvugt/simpleapp --name=simple-app**
- **oc get is -n openshift**
- **oc get builds**: allows for monitoring the build process
- **oc get buildconfig**: shows the buildconfig used
- **oc get deployment**: shows the resulting deployment

Using Blue/Green Deployments in Kubernetes and OpenShift

Understanding Blue/Green Deployments

- A blue/green deployment is a way of accomplishing a zero-downtime application upgrade
- The blue deployment is the current application
- The green deployment is the new application
- Once the green deployment is ready, traffic is re-routed to the new application version
- Kubernetes Deployment and Service resources make implementing blue/green deployment in Kubernetes easy

Procedure Overview

- Notice this can be done in multiple ways
- Start with already running deployment and service
- Create new deployment running the new version
- Perform a health check
- If health check passes, update the load balancer and remove old deployment
- if health check fails, stop

Detailed Procedure

- **oc create deployment blue-nginx --image=bitnami/nginx:1.14 --replicas=3**
- **oc expose deployment blue-nginx --port=80 --name=bgnginx**
- **oc get deploy blue-nginx -o yaml > green-nginx.yaml**
  - Change Image version
  - Change "blue" to "green" throughout
- **oc create -f green-nginx.yaml**
- **oc get pods**
- **oc delete svc bgnginx**
- **oc expose deployment green-nginx --port=80 --name=bgnginx**
- **oc delete deployment blue-nginx**

Container Devops in 4 Weeks

Understanding Helm Charts, Operators and Custom Resources

What is this about?

- It's all about running custom applications in Kubernetes
- Helm Charts are like packages that can be used in Kubernetes
- Custom Resource Definitions allow you to extend the Kubernetes API to add new resources (discussed last week)
- Operators are using Custom Resource Definitions to provide applications

Understanding Helm

- Helm is about reusing YAML manifests through templates
- These templates work with properties that are defined in a separate file
- Helm merges the YAML templates with the values before applying them to the cluster
- The resulting package is called a Helm Chart

Understanding Operators

- An operator extends the Kubernetes API to run a stateful application to run natively on Kubernetes
- An operator consists of Kubernetes custom resources and/or APIs and controllers
- Operators are typically written in a standard programming language like Golang, Python or Java
- Operators are packages as container images and deployed using YAML manifests
- As a result, new resources will be available in the cluster
- Operators can be distributed using Helm Charts
- We'll later use operators to deploy Red Hat CI/CD in OpenShift

Working with Helm

- To work with Helm, you'll need to install it
- Make sure you use Helm 3, version 2 is obsolete
- Helm charts are the helm packages
- A running instance of a helm chart is called a release

Installing Helm

- Apply instructions on https://docs.openshift.com/container-platform/4.6/cli_reference/helm_cli/getting-started-with-helm-on-openshift-container-platform.html
- Use **helm version** to verify
- Use **helm create my-demo-app** and check the directory that is created and its contents

Demo: Installing a Helm Chart on OpenShift

- oc new-project mysql
- helm repo add stable https://charts.helm.sh/stable
- helm repo update
- helm list
- helm install example-mysql stable/mysql
- helm list
- oc get all

Demo: Working with Customized Helm Charts

- **cat my-ghost-app/Chart.yaml**
- **cat my-ghost-app/templates/deployment.yaml**
- **cat my-ghost-app/templates/service.yaml**
- **cat my-ghost-app/values.yaml**
- **helm template --debug my-ghost-app**
- **helm install -f my-ghost-app/values.yaml my-ghost-app my-ghost-app/**

Container Devops in 4 Weeks

Summary: Container Based Devops

Summary

- Kubernetes and OpenShift are awesome tools for DevOps
- The CI/CD part is filled in by working with container images that are easily updated
- The MicroServices approach is implemented by using multiple containers and connect these using variables
- The variables are easily decoupled using Kubernetes ConfigMaps and Secrets
- DevOps deployment strategies such as Blue/green and Canary deployment are easily implemented using Kubernetes
- OpenShift is adding S2I to make the CI/CD part even easier

Further Learning

Related Live Courses

- Containers:
  - Containers in 4 Hours
- Kubernetes
  - Kubernetes in 4 Hours
  - CKAD Crash Course
  - CKA Crash Course
  - Building Microservices with Containers
- OpenShift
  - Getting Started with OpenShift
  - EX180 Crash Course
  - EX280 Crash Course

Related Recorded Courses

- Getting Started with Kubernetes, 2nd Edition
- Hands-on Kubernetes
- Certified Kubernetes Application Developer
- Certified Kubernetes Administrator, 2nd Edition
- Red Hat OpenShift Administration: Red Hat EX280
- Modern Container-Based DevOps: Managing Microservices using Kubernetes and Docker